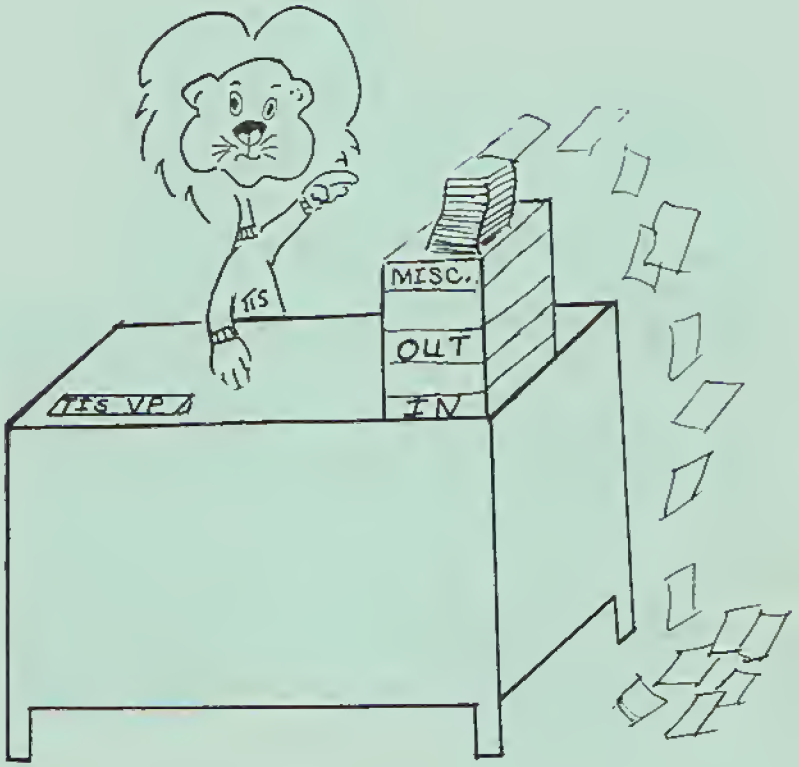
TIS Workbook 5

TOTAL INFORMATION SERVICES



PET Miscellaneous

TABLE OF CONTENTS

I. INTRODUCTION

This workbook gives a series of exercises that show the user how to use some of the miscellaneous features of the Commodore PET 2001 computer. The most effective way to use the workbook is to sit down with a PET and go through the exercises as they are presented. Enough space has been provided in the workbook for you to add your own examples as you develop them. Later, when you need to refresh your memory on a particular topic, these examples should supply pertinent, meaningful information.

A. Assumptions Made About the User

Some PET users are comfortable with mathematics. However, this workbook assumes that the majority are not. For that reason, most exercises will use nothing more than high school arithmetic.

This workbook also assumes you know something about the syntax and semantics of the BASIC programming language. If you do not, we recommend that you obtain one of the following books:

BASIC Programming, J. Kemeny and T. Kurtz
BASIC, Albrecht, Finke, Brown
What Do You Do After You Hit Return? Peoples Computer Co.
Basic BASIC, James Coan
Advanced BASIC, James Coan

Read about BASIC syntax; then alternate between this workbook and your text on BASIC. That way you will learn both BASIC and how to use the PET.

B. Notation Used in This Workbook

We use a consistent notation in this workbook to indicate what is to be typed on the keyboard (T:), what appears on the TV display (R:), and what indicates blanks are to be typed (b). For example:

T: info ('RETURN') key

means to type the characters contained on the line after the colon (:) followed by a 'RETURN'.

R: response

means that the system response to the previous T: line should be a line on the TV.

Blanks are important. They are specified by b. For example:

T: ?"ABbC" ('RETURN' key)

means type ?, then ", then the letter A, then the letter B, then a space, then the letter C, then " followed by a 'RETURN'. Now let's run that example all together:

T: ?"ABbC" ('RETURN' key)
R: AB C

The special keys on the PET keyboard can cause some confusion. This workbook identifies the special keys with the notation 'KEYNAME'. 'KEYNAME' means press the named key. The unquoted sequence of characters OTHER means, press the five keys O T H E R in succession.

Example: Special key notation

T: 'STOP'

means press the key labeled STOP.

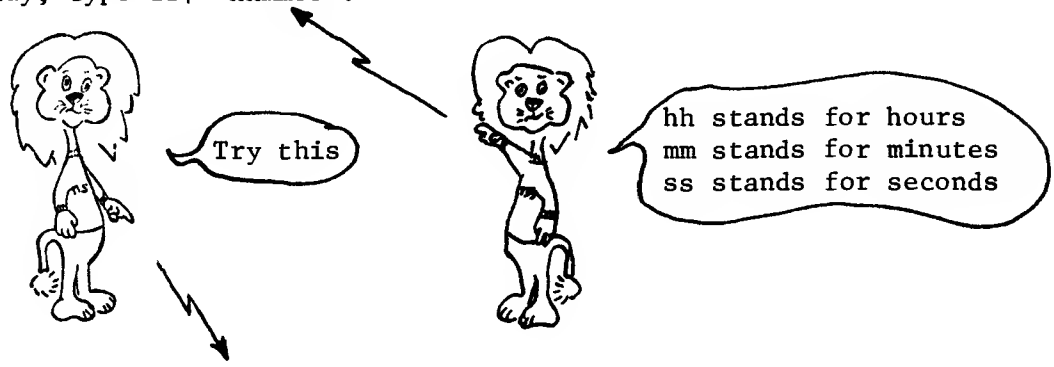
T: STOP
R: BREAK
R: READY.

means press the four keys S T O P in succession.

II. THE PET INTERNAL CLOCK

A. The TI$ Variable

This variable is used to set the internal clock to hours (0-24), minutes and seconds. Each time the power is turned on, TI$ is set to 000000. TI$ is then incremented by one each second. The rightmost two characters are seconds, the middle two characters are minutes, and the leftmost two characters are hours. By printing TI$ at the end of a session you can tell how long the power has been on. If you want to set TI$ to the real time of day, type TI$="hhmmss".



Exercise: Enter the time of day and display it.

```
T:  NEW
T:  10 TI$="141000"
T:  20 PRINT CHR$(147)
T:  30 PRINT CHR$(145);TI$:GETA$:IFA$=""GO TO 30
T:  RUN
R:  141000
```

The CHR$(147) is used to clear the screen. The CHR$(145) is for the up cursor so that the time will always be printed on the same line. When you try this exercise, you will notice that the response is changing every second. To stop the code, press any key.

The clock (TI$) may also be set using string variables under program control.

Exercise: Set TI$ under program control.

```
T:  NEW
T:  10 PRINT"ENTER THE TIME AS HH,MM,SS"
T:  20 INPUT HR$,MN$,SC$
T:  30 PRINT CHR$(147)
T:  40 TI$=HR$+MN$+SC$
```

This sets TI$ to the time of day that you entered.

```
T:  50 HR$=LEFT$(TI$,2)
T:  60 MN$=MID$(TI$,3,2)
T:  70 SC$=RIGHT$(TI$,2)
```

Splits the time in TI$ into hours, minutes, and seconds.

```
T:  80 PRINT CHR$(145);HR$;":";MN$;":";SC$
T:  90 GET A$:IF A$=""GO TO 50
```

Print the current time and branch for a new update of the display.

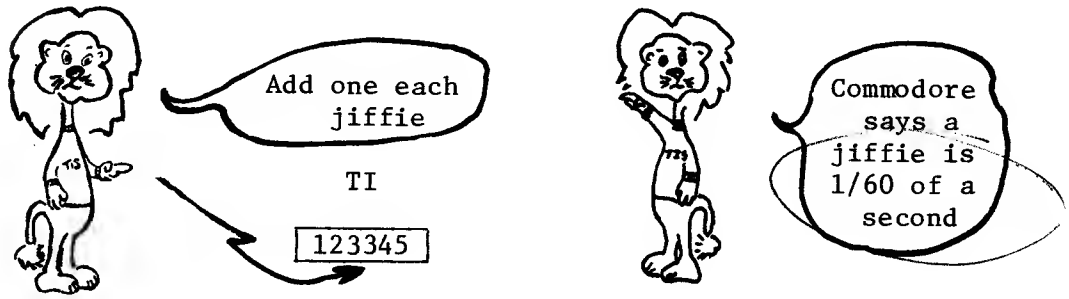
```
T:  100 END
T:  RUN
R:  ENTER THE TIME AS HH,MM,SS
R:  ?
```

The response will change every second. As you have probably noticed by now, it is hard to show that on paper. Press any key to stop execution. If you get the message ?ILLEGAL QUANTITY ERROR IN 40, RUN the program again and make sure that you enter a legal time and separate hours from minutes and minutes from seconds with commas.

The PET clock automatically rolls over at midnight. Enter 23, 59, 50. In 10 seconds the clock should change to 00:00:00.

B. The TI Variable

The TI variable holds the count of the number of jiffies since the power was turned on. It is incremented by one each every jiffie.



TI can not be set in the same manner TI$ can. It can be set to zero, however, by setting TI$ to zero. It is very useful when building in small delays in a program. It can also be used as a reaction timer and other real time uses.

Exercise: Display both TI and TI$.

```
T:  NEW
T:  10 PRINT CHR$(145),TI,TI$
T:  20 GET A$:IF A$=""GO TO 10
T:  RUN
R:  115120 000412
```

(These numbers depend on how long since you turned your PET on).

The clock is a useful tool in many applications. It may be used to keep track of the number of hours your PET is used. It may be used to build in delays in your programs or to set a time limit in a game or other application that requires a response in a given time. The following subroutine may be used to set a time limit in a game such as chess. You may enter whatever time you want to start from. Generally, this would be 00,00,00 or the wall clock time. You may then enter the maximum time limit before a response is expected. After the initial set up, this routine should be entered at statement 80.

```
T:  NEW
T:  10 PRINT"ENTER THE TIME IN THE FORM HH,MM,SS"
T:  20 INPUT H$,M$,S$
T:  30 PRINT"ENTER THE TIME LIMIT IN THE FORM HH,MM,SS"
T:  40 INPUT HR$,MN$,SC$
T:  50 TM$=HR$+MN$+SC$
T:  60 PRINT"'CLR'"
T:  70 PRINT"'HOME'  "
T:  80 TI$=H$+M$+S$
T:  90 GOSUB 60000
T:  100 GO TO 90
```

This is to set up the initial conditions. If you want the time to be reinitialized, that is, reset to the original value (all zeros for example) after each "play", then remove statement 80, add 59090 TI$=H$+M$+S$ and change statement 90 to 90 GOSUB 59090.

```
T:  60000 B=VAL(TM$)+VAL(TI$)
T:  60010 PRINT"'HOME'  "
```

Set B equal to the maximum time allowed and move the cursor to home.

```
T:  60020 A$=LEFT$(TI$,2)
T:  60030 A1$=MID$(TI$,3,2)
T:  60040 A2$=RIGHT$(TI$,2)
T:  60050 PRINT A$;":";A1$;":";A2$
```

Get the current time step and print it.

```
T:  60060 IF B-VAL(TI$) < =0 GO TO 60090
```

Check to see if the time is up.

```
T:  60070 GET C$:IFC$=""GO TO 60010
```

Check to see if there is a response from the keyboard. If there is not a response, update the time step.

```
T:  60080 RETURN
```

If you got a response within the time limit, then return to the calling program.

```
T:  60090 PRINT"TIMES UP"
T:  60100 GET C$:IFC$=""GO TO 60100
T:  60106 PRINT"'HOME'"
```

If there wasn't a response within the time interval, then wait for a key to by typed before starting another time interval.

```
T:  60110 RETURN

T:  RUN
R:  ENTER THE TIME IN THE FORM HH,MM,SS
R:  ?
T:  00,00,00
R:  ENTER THE TIME LIMIT IN THE FORM HH,MM,SS
R:  ?
T:  00,00,03
R:  00:00:01
R:  00:00:02
R:  00:00:03
R:  TIMES UP
```

If you hit any key before the three seconds are up, a new three-second count will begin.

Exercise: POKE the heart character on the screen and use TI to make it blink every 20 jiffies, i.e., every one-third of a second.

```
T:  NEW
T:  PRINT"'CLR'"
T:  20 TC=TI+20
T:  30 POKE 33020,83: POKE 33021,32
```

Set the time interval for 20 jiffies and display a heart (83) and a blank (32).

```
T:  40 IF TI < TC GO TO 40
```

Wait for 20 jiffies.

```
T:  50 TC=TI+20
T:  60 POKE 33020,32:POKE 33021,83
```

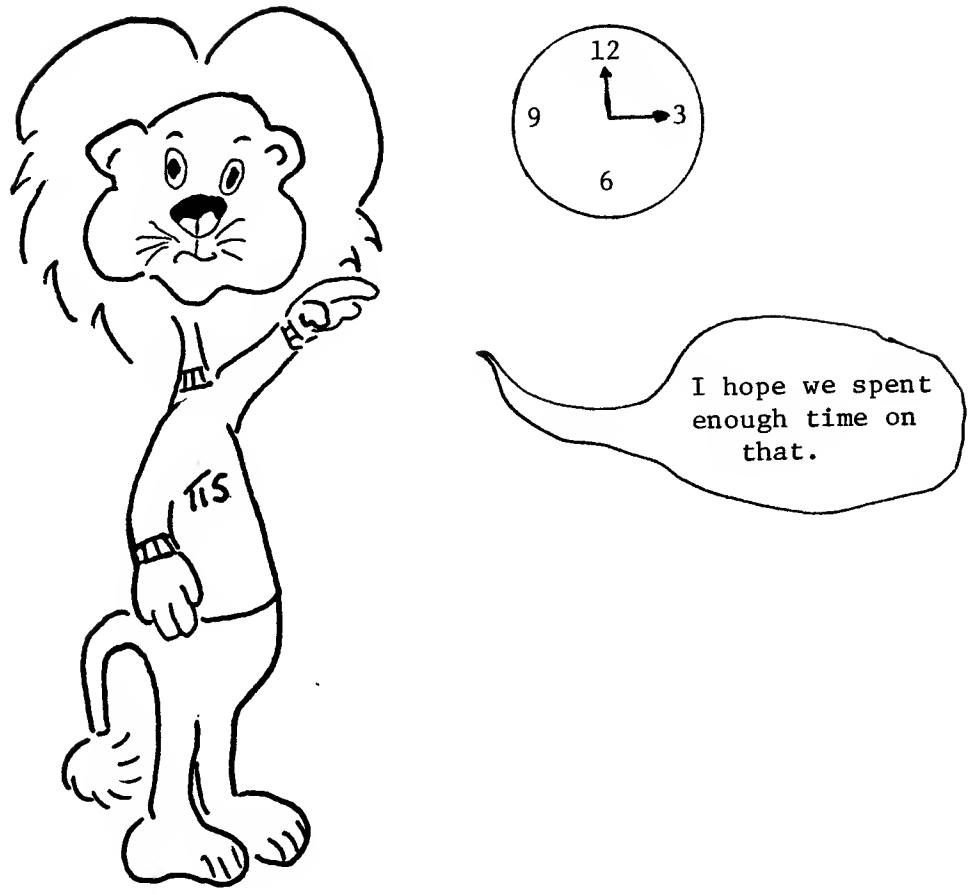
Set a new time interval and replace the previous heart with a blank. Replace the previous blank with a heart.

```
T:  70 IF TI < TC GO TO 70
```

Wait for 20 jiffies.

```
T:  80 GO TO 20
T:  RUN
```

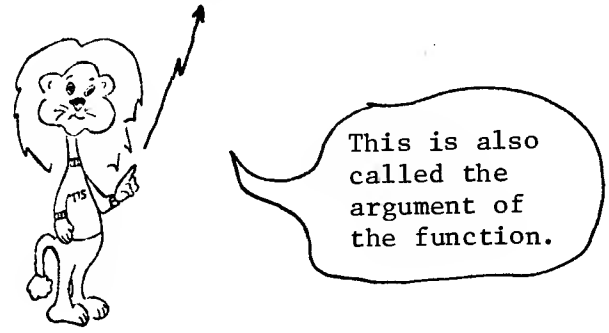
This program will move the heart character from one position to another so that it appears to blink off and on. Notice that a small amount of "snow" appears on the screen as the heart is blinked. Snow is random pieces of unwanted white spots. See Section IX for a program that updates the screen without snow.


12
9
3
6
I hope we spent enough time on that.
TIS

III. USER DEFINED FUNCTIONS

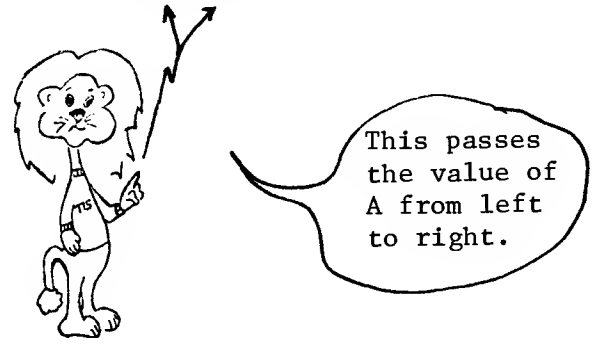
The DEF FN function allows you do define a one-statement function that may later be used in your program like the system functions.

The general form is DEF FN name (variable) = some arithmetic statement.



The value of a variable may be passed to the right side of the function by using the variable as the argument in the function call. What this means is that we can pass any value, for example, 2, to the equation on the right by setting A=2, then use A in the equation. Let's define a user function called CUBE that will take the value of the variable A and cube it. In order to do this we must use A on the right side of the equal sign.

```
DEF FN CUBE(A)=A*A*A
```



Functions are convenient when you have a long arithmetic statement that is to be solved in several places in your program. You can define the statement once and then call the function whenever you need a result.

Exercise: Define a user function to calculate the volume of a cylinder.

```
T:  NEW
T:  10 DEF FNVOL(R)=H*π *R ↑ 2
T:  20 H=10
T:  30 R=2
T:  40 V=FNVOL(R)
```

The formula for the volume of a cylinder is the height times times the radius squared. V is the volume of a cylinder of height 10 and radius 2.

```
T:  50 NR=H/2
T?  60 VI=FNVOL(NR)
```

VI is the volume of a cylinder of height 10 and radius equal to one-half the height. We calculated a new radius (NR) and passed that value to the arithmetic statement function defined in statement 10 by inserting NR in the function call FNVOL(NR).

```
T:  70 PRINT V;VI
T:  RUN
R:  125.663706  785.398164
R:  READY
```

The function name must be less than five alphabetic characters or a syntax error occurs; however, only the first two characters after FN are actually used by the system.

Exercise: Show that the first two characters of a function name are all that are used.

```
T:  NEW
T:  10 DEF FNCUBE(A)=A*A*A
T:  20 A=2
T:  30 J=FNCU(A)
T:  40 PRINT J
T:  RUN
E:  8
R:  READY
```

You may use more than five characters in a function name if a number is used before the fifth position in the name. For example: DEF FN ABC2DEF(K)=A+5 will work without getting a syntax error.

IV. THE RANDOM NUMBER GENERATOR

Random number generators are used in many applications such as games (shuffling cards, rolling dice, etc.), statistical sampling (choosing the people to talk to), and other areas where you need to select things or events in an unbiased way. Random number generators used in computers use a "seed" to start generating a sequence of random numbers. A seed is nothing more than a number that tells the random number generator where to start. There are many different methods used to generate random numbers and many different theories concerning these methods. One method is to take the seed and multiply it by itself and keep some number of low order bits, then multiply these bits and so on. During program execution, the last number generated is stored so that each time the random number generator is called, it can use this number to continue the sequence. Obviously, 0 and 1 would have to be handled as special cases in the above method. Each time the random number generator is started after the power is turned on, it will generate the same sequence of numbers given the same seed to start with. Try this by:

```
T:  NEW
T:  10 A=RND(2)
T:  20 PRINT A
T:  30 C=C+1
T:  40 IF C < 5 GO TO 10
T:  RUN
```

Now write down the five numbers generated, then turn your PET OFF and BACK ON and RUN the same program. You should get the same sequence of numbers.

The RND function generates numbers in the range $0 <= R < 1$. The general form is R=RND(X) when R is a variable and X is a seed that the RND function uses to determine the starting sequence of numbers generated. If X is negative, RND(X) will generate the same number each time RND(X) is called. A different number is generated for each negative value of X. If X is positive, a new random sequence is given each time RND(X) is called. If X is 0, one of our PETs gives one every time RND(X) is called. The other PET gives a non-repeatable sequence of about ten unique numbers. The sequence will begin with either .588, .619, or .998 each time RND(X) is called.

Exercise: Determine the random number generated for the negative numbers -1 through -10.

```
T:  10 FOR I=-10 TO -1
T:  20 A=RND(I)
T:  30 PRINT A;I
T:  40 NEXT I
T:  50 GO TO 10
T:  RUN
R:  3.73729563E-    -10
R:  3.3647666E-08 -9
R:  2.99223757E-08 -8
R:  5.2273208E-08 -7
R:  4.48226274E-08 -6
R:  3.73720468E-08 -5
R:  2.99214662E-08 -4
R:  4.48217179E-08 -3
R:  2.99205567E-08 -2
R:  2.99196472E-08 -1
```

This sequence will continue to repeat until the 'STOP' key is typed. This shows that the same number is generated each time the same negative number is given to RND.

Often it is desirable to generate a repeatable sequence of numbers before generating a truly random sequence. This is very useful when you are trying to debug a program.

Exercise: Generate a repeatable sequence of "random" numbers.

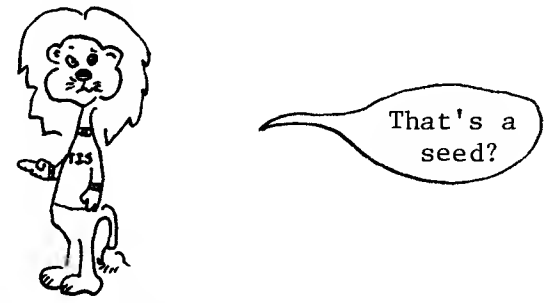
```
T:  NEW
T:  10 FOR I=1 TO 10
T:  20 A=-RND(-I)
T:  30 B=RND(A)
T:  40 PRINT B
T:  50 NEXT I
T:  RUN
R:
R:
     .
     .
     .
R:
```

Giving the RND function the same sequence of negative numbers each time the function is called generates a repeatable sequence of numbers such that 0<=B<1. If you wish to change the sequence that is generated, just change the seed to some other value.

The internal clock is an excellent source for seeds to be given to the RND function. The seeds are random in the sense that you can not predict what number will be chosen. However, they are not completely random since each number does not have an equal probability of being chosen.

Exercise: Use the clock for the seed in RND.

```
T:  NEW
T:  10 FOR I=1 TO 20
T:  20 A=RND(TI)
T:  30 PRINT A
T:  40 NEXT I
T:  RUN
```



This produces a random sequence of numbers such that 0<=A<1. If you wish to generate a sequence of random integers, say between 1 and 13, change statement 20.

```
T:  20 A%=13*RND(TI)+1
```

Multiplying by 13 gives a number X from 0<=X<13, adding 1 gives a number X from 1<=X<14. A% will only take the integers in 1<=X<14. This generates a sequence of integers X such that 1<=X<13. You can use this to pick a card, for example.

```
T:  30 PRINT A%
T:  RUN
R:
```

If you wish to shuffle a deck of cards you would use the number 52 instead of 13.

V. USING MACHINE CODE FROM BASIC

The SYS(X) and USR(X) statements allow you to go from a BASIC program to a machine language program and back again. Sometimes it is necessary to use the 6502 machine language to have your PET do things that you would find difficult if not impossible to do in BASIC. For example, you can write programs that take less memory and execute faster than they would in BASIC. This is normally not necessary unless you have very long calculations or are very short on memory. When coding in machine language for the 6502 processor, memory is thought of in terms of bytes and pages. Each location is specified by giving the low order address (byte) first and then giving the high order address (page) second. Each page contains 256 bytes. See Fig. 5-1.

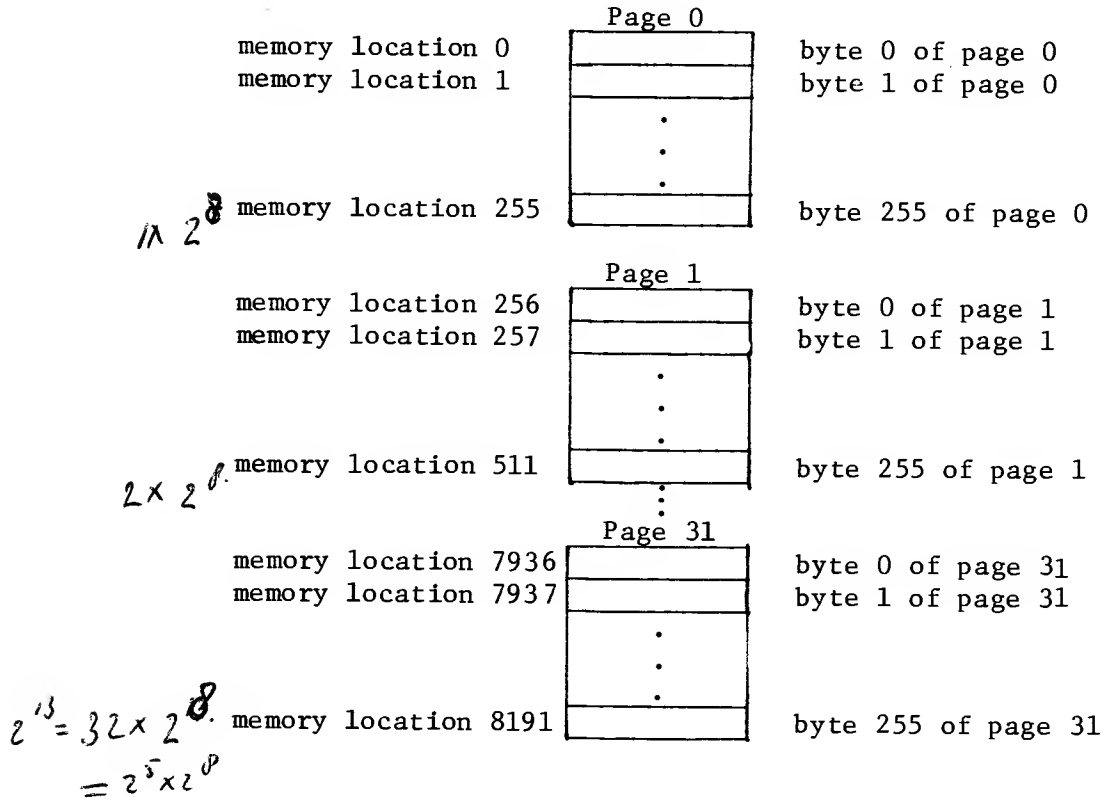
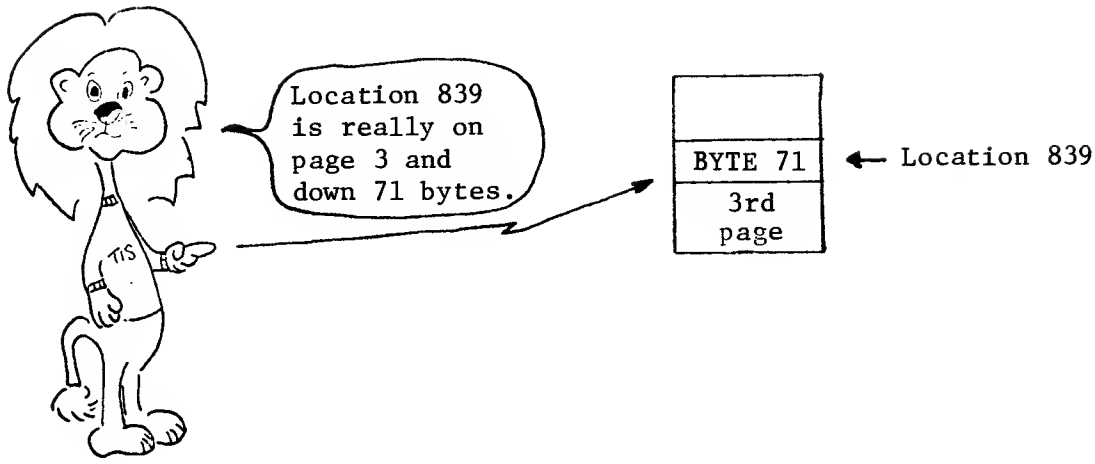


Fig. 5-1.

For example, to specify the address of location 839 we first find the page number by dividing 839 by 256. 839/256=3 with a remainder of 71. The quotient, 3, is the page number, while the remainder, 71, is the byte number.



A. The SYS(X) Command

Control of the PET is transferred to the decimal address at X. To show the use of the SYS command we will write a simple machine language code to add the contents of two memory locations and store the result in a third location. We will use the buffer area reserved for the second cassette to store our code. This buffer area is the 191 bytes from location 826 to location 1017.

Exercise: Run a machine language program using the SYS command.

```
T:  NEW
T:  10 POKE 839,1
T:  20 POKE 840,1
```

Put the data value 1 in locations 839 and 840.

```
T:  25 POKE 826,24          :REM  CLC
T:  30 POKE 827,173         :REM  LDA
T:  40 POKE 828,71          :REM  3*256+71=839(ADDRESS)
T:  50 POKE 829,3
```

24 is the decimal equivalent of the op code CLC - clear carry flag. 173 is the decimal equivalent of the op code LDA - load the accumulator from memory. 71 is the low order byte of memory address 839. 3 is the high order byte (the page number) of memory address 839. Together these two bytes tell the LDA instruction where to go in memory to get the data to load.

```
T:  60 POKE 830,109                    :REM  ADC
T:  70 POKE 831,72
T:  80 POKE 832,3                      :REM  3*256+72=840(ADDRESS)
```

109 is the decimal equivalent of the op code ADC - add memory to accumulator with carry. 72 and 3 are the byte and page number of location 840.

```
T:  90 POKE 833,141                    :REM  STA
T:  100 POKE 834,244
T:  110 POKE 835,129                   :REM  129*256+244=33268(ADDRESS)
```

141 is the decimal equivalent of the op code STA - store accumulator in memory. 244 and 129 are the bytes and page number of location 33268 which is the center of the TV screen. When this code is run, a B should appear there. The character B is displayed because a 2 is the numerical code for a B.

```
T:  120 POKE 836,96                    :REM  RTS
```

96 is the decimal equivalent of the op code RTS - return from subroutine.

```
T:  130 PRINT CHR$(147)
T:  RUN
R:  READY
T:  SYS(826)
R:  READY
R:                          B
T:  PRINT PEEK (33268)
R:  2
```

The POKE statement is usedto enter the machine instruction in memory and the SYS(X) command to branch (transfer control) to location X and begin execution. The SYS(X) command is similar to the RUN command in that execution begins as soon as the 'RETURN' key is typed. The SYS command can also be used in the program.

```
T:  140 SYS(826)
T:  RUN
```

You should get the same results as before.

B. The USR(X) Function

The USR(X) function transfers control of the PET to the location that is stored in memory locations 1 and 2.

Let's modify the previous program to use USR. We can branch to a machine language subroutine and return to continue execution in the main program.

```
T:  140 POKE 1,58
T:  150 POKE 2,3
T:  160 K=7
T:  170 A=USR(K)
T:  180 PRINT A,K
T:  RUN
R:  7        7
R:               B
T:  PRINT PEEK (33268)
R:  2
```

In locations 1 and 2 put the address of the subroutine that you want to execute (location 826). Remember that location 826 is byte 58 of page 3. 826/256=3 with a remainder of 58. When the USR statement was executed, control was transferred to the instructions starting at location 826. USR finds the address in locations 1 and 2. The argument is passed to the machine language subroutine by placing the value of the argument in the floating point accumulator. A new value may be returned to the calling routine by placing that value in the floating point accumulator before the RTS (return to subroutine) instruction is executed. The floating point accumulator is located at locations 176-180 decimal (00B0-00B4 hex).

The floating point accumulator is in the following format:

Address		Byte	
Decimal	Hex		
176	00B0	Exponent +128	
177	00B1	Fraction	Most significant bit
178	00B2	Fraction	
179	00B3	Fraction	
180	00B4	Fraction	Least significant bit
181	00B5	Sign of Fraction	High order bit is 0 if the number is 0 or positive and 1 if the number is negative

The floating point number is always normalized, that is, the fractional part is shifted until the most significant bit is in position 7 of location 177. Position 7 is the high order bit of the byte. The low order bit is in position 0. The radix point* is always assumed to be to the left of bit 7 in location 177.

Let's look at a few examples of some floating point numbers. Let's see how the number 2 is represented in the floating point accumulator. First convert 2.0 to a normalized binary number.

$$2=010_2$$

$$2=.2 \times 10_{10} \qquad 010_2 = .1 \times 2\uparrow 2$$

Both numbers have been normalized by shifting the radix point to the left and then multiplying that number by the base raised to a power. In the first case the base is ten and the power is one (the number of places to the left the radix point was moved). In the second case the base is two and the power is 2 (again the number of places the radix point was moved).

Now that you have normalized binary number, let's see how this number is represented in the floating point accumulator.

In location 176 is the exponent + 128. The exponent is 2 so in loaction 176 is the number 130. In location 177 is the number 128. How did 128 get in location 177? Remember that the number 2 was normalized which places a one bit in position 7 of location 177.

Locations 178-181 all contain 0.

Now let's look at the actual bit patterns.

LOCATION 176	LOCATION 177	LOCATION 178
10000010 76543210	10000000 76543210	00000000 76543210
130	128	0

LOCATION 179	LOCATION 180	LOCATION 181
00000000 76543210	00000000 76543210	00000000 76543210
0	0	0

*Radix point is synonymous with base point, i.e., decimal point, binary point, etc.

What does a -2 look like

LOCATION 176	LOCATION 177	LOCATION 178
10000010 76543210	10000000 76543210	00000000 76543210
130	128	0

LOCATION 179	LOCATION 180	LOCATION 181
00000000 76543210	00000000 76543210	10000000 76543210
0	0	128

The number 128 in location 181 says that you have a negative number. Location 181 will contain a 0 if the number is 0 or positive and a 1 in bit position 7 i.e., 128 if the number is negative.

How would a number less than 1 be represented?

The number .5 is represented as: $.5 = 2^{-1}$

LOCATION 176	LOCATION 177	LOCATION 178
01111111 76543210	10000000 76543210	00000000 76543210
127	128	0

LOCATION 179	LOCATION 180	LOCATION 181
00000000 76543210	00000000 76543210	00000000 76543210
0	0	0

The number 0 is represented as:

LOCATION 176	LOCATION 177	LOCATION 178
00000000 76543210	10000000 76543210	00000000 76543210
0	128	0

LOCATION 179	LOCATION 180	LOCATION 181
0	0	0

Since the PEEK statement uses the floating point accumulator you can not dirrectly look at the numbers in location 176-181. Let's write a machine language program that stores the contents of the floating point accumulator in locations 876-881 and then look at these locations. The POKE statements load memory with the machine language code that will be executed when called by the USR statement.

```
T:  NEW
T:  10 REM                                  MNEMONIC     OP CODE
T:  15 REM                                               (HEX)
T:  20 POKE 1,77
T:  30 POKE 2,3

T:  340 POKE 845,160                        :REM  LDY  A0
T:  350 POKE 846,6
```

Set up to load index register Y with the number 6.

```
T:  360 POKE 847,185                        :REM  LDA  B9
T:  380 POKE 848,175
T:  390 POKE 849,0
```

Set up to load the accumulator with the contents of location 175 modified by index Y (This will load the floating point accumulator)

```
T:  400 POKE 850,153                        :REM  STA  99
T:  410 POKE 851, 107
T:  420 POKE 852,3
```

Set up to store the accumulator in location 875 modified by register Y.

```
T:  430 POKE 853,136                        :REM  DEY  88
```

Decrement index register Y by one.

```
T:  440 POKE 854,208                        :REM  BNE  D0
T:  450 POKE 855,247
```

Set up to branch back to location 847 if Y is not zero. In order to determine the relative address of location 847 begin with the BNE instruction and count backward eight instructions to the LDA instruction. Convert 8 to hex and subtract from FF. This number (F7) converted to decimal (247) is the relative address of the LDA instruction.

```
T:  460 POKE 856,96                         :REM  RTS  60
```

Return from the machine language subroutine.

```
T:  490 FOR I=-5 TO 5
T:  500 J= USR(I)
T:  510 PRINT "J=";J;" I=";I
```

Look at the numbers between -5 and 5

```
T:  520 FOR K=1 TO 6
T:  530 PRINT PEEK (875+K);
T:  540 NEXT K
```

Look at what was in the floating point accumulator.

```
T:  550 PRINT
T:  560 NEXT I
T:  563 END
T:  RUN
R:  J=-5  I=-5
R:  131  160  0  0  0  160
 .
 .
 .
R:  J=0  I=0
R:  0  128  0  0  0  0
 .
 .
 .
```

To check some numbers less than 1

```
T:  490 FOR I=-.5 TO .5 STEP.1
T:  RUN
R:  J=-.5  I=-.5
R:  128  128  0  0  0  128
 .
 .
 .
R:  J=.1  I=.1
R:  125  204  204  204  203  76
 .
 .
 .
R:  J=.5  I=.5
R:  127  255  255  255  255  127
```

Now let's change the value of the variable (I) in the machine language portion of the code so that J will be different from I. We will simply change the sign of the value returned in the USR call.

```
T:  451 POKE 856,169                    :REM  LDA  A9
T:  452 POKE 857,128
```

Set up to load the accumulator with the number 128. This turns on the bit in position 7 .

```
T:  453 POKE 858,69                    :REM  EOR  45
T:  454 POKE 859,181
```

Set up to do an exclusive OR with the sign byte (loc. 181) of the floating point accumulator.

```
T:  455 POKE 859,133                   :REM  STA  85
T:  456 POKE 860,181
```

Set up to store the result in location 181.

```
T:  460 POKE 861,96                    :REM  RTS  60
```

Return from the machine language subroutine.

```
T:  RUN
```

These examples show how to use the USR function and pass a number back and forth between BASIC and a machine language program.

VI. EXECUTION OF COMMANDS UNDER PROGRAM CONTROL

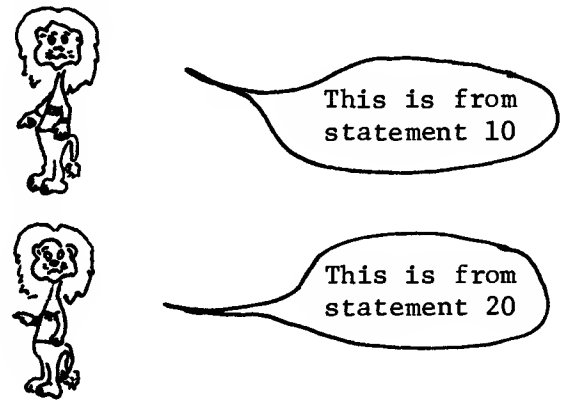
The BASIC commands may be executed under program control, that is, as part of your program. This can be a versatile and powerful tool with some commands. Those commands that return control to the system after they are executed are in general less powerful within a program than those that return control to the program.

A. LIST

A program can execute the list command to list all or part of itself. The list command is of limited use because after the command is executed, control is returned to the system. This means that no more statements are executed without user intervention.

Exercise: Using the LIST command under program control.

```
T:  NEW
T:  10 PRINT"LIST 1"
T"  20 LIST
T:  30 PRINT"LIST 2"
T:  RUN
R:  LIST 1
R:  10 PRINT"LIST 1"
R:  20 LIST
R:  30 PRINT"LIST 2"
R:  READY.
```



Notice that statement 30 was not executed.

B. RUN

Exercise: Using the RUN command under program control.

```
T:  NEW
T:  10 FOR I=1 TO 5
T:  20 A=A+1
T:  30 IF A > 100 THEN STOP
T:  40 NEXT I
T:  50 PRINT A
T:  60 RUN
T:  RUN
```

What would you expect to get?

The RUN command reinitializes the variables before execution begins. Therefore the above program will continually print 5's.

C. LOAD

One of the most interesting commands to use under program control is the LOAD command. Using LOAD under program control allows you to overlay programs in memory. You can execute very large programs by breaking them up and putting them on cassette tape as small routines and then loading and executing these routines one at a time.

Exercise: Running multiple programs (overlays) from cassette tape. Rewind a scratch program tape.

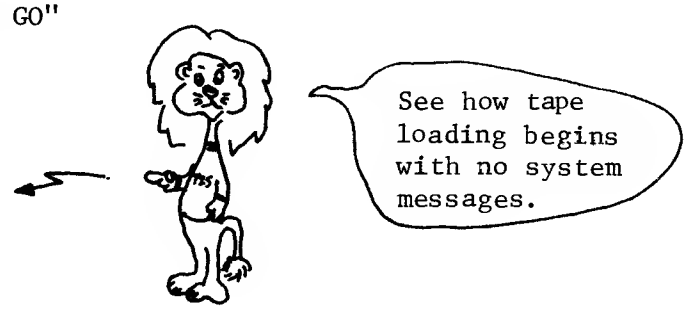
```
T:  NEW
T:  10 PRINT"THIS IS PROGRAM 1 FROM TAPE"
T:  20 LOAD"PRGM2"
T:  SAVE"PRGM1",1,2
R:  PRESS PLAY & RECORD ON TAPE #1
R:  OK
R:  WRITING PRGM1
R:  READY.
```

Rewind the cassette.

```
T:  VERIFY
R:  PRESS PLAY ON TAPE #1
R:  OK
R:  SEARCHING
R:  FOUND PRGM1
R:  VERIFYING
R:  OK
T:  NEW
T:  10 PRINT"THIS IS PROGRAM 2 FROM TAPE"
T:  SAVE"PRGM2",1,2
R:  PRESS PLAY & RECORD ON TAPE #1
R:  OK
R:  WRITING PRGM2
R:  READY.
T:  NEW
```

You now have two programs on the cassette tape. Rewind the cassette and we will write a program that will execute, then call PRGM1. It will execute, and then call PRGM2, which will also execute.

```
T:  10 PRINT CHR$(147);"HERE WE GO"
T:  20 LOAD"PRGM1"
T:  RUN
R:  HERE WE GO
R:  PRESS PLAY ON TAPE #1
R:  OK
R:  THIS IS PROGRAM 1 FROM TAPE
R:  THIS IS PROGRAM 2 FROM TAPE
R:  READY.
```



As you can see, we executed three different programs each of which had all of memory available. Except for the first program, each of the others replaced the program that called it. Notice that the normal message, "SEARCHING FOR PRGM1" is not displayed. Notice also that the loaded program begins execution immediately, that is, no need to type RUN.

When overlaying programs, be sure that the last program on tape has an EOT written after it so that if the file is not found, the tape will not run away. Remember, an EOT may be written by typing SAVE "filename",1,2 when you SAVE the file on tape.

D. SAVE

The SAVE command could be the last line in your program so that each time you modify your code and RUN it, you would automatically be reminded to save the latest version.

```
T:  NEW
T:  10 PRINT"SAVE"
T:  20 SAVE"SAVE",1,2
T:  30 PRINT "AFTER SAVE"
T:  RUN
R:  SAVE
R:  PRESS PLAY & RECORD ON TAPE #1
R:  OK
R:  AFTER SAVE
R:  READY .
```

E. CLR

The CLR command sets all numeric variables to 0 and all string variables to ""(null). CLR does not zero any program statements; however, it can be used to redimension arrays during program execution. Caution must be exercised because all variables are either set to 0 or null. The CLR command is different from the 'CLR' key. The 'CLR' key blanks the TV screen but does not change variables or statements in memory.

Exercise: Redimension arrays during execution.

```
T:  NEW
T:  10 DIM A(10)
T:  20 FOR I=1 TO 10
T:  30 A(I)=I
T:  40 PRINT A(I)
T:  50 NEXT I
T:  60 PRINT"I=";I
T:  70 DIM A(20)
T:  80 FOR I=1 TO 20
T:  90 A(I)=I*I
T:  100 PRINT A(I)
T:  110 NEXT I
T:  120 PRINT"I=";I
T:  RUN
R:  1
R:  2
    .
    .
    .
R:  10
R:  I=11
R:  ? REDIM'D ARRAY ERROR IN 70
R:  READY .
```

Now let us add a CLR statement to the above program.

```
T:  55 CLR
T:  RUN
R:  1
R:  2
    .
    .
    .
R:  10
R:  I=0
R:  1
R:  4
    .
    .
    .
R:  400
R:  I=21
R:  READY.
```

CLR will allow a DIM statement to change the size of an array during execution, but remember all the rest of the variables are reset also.

VII. SPACE AND POSITION INFORMATION

The FRE(D) and POS(D) functions give the status of the PET. FRE(D) returns the available memory. POS(D) returns the position of the cursor (the next available print position).

A. FRE(D)

The FRE(D) function is used to determine the number of bytes of memory that are available for use. The argument is a dummy one (it does not do anything but just sit there; however, you get a syntax error if it is left out). The argument may be either a variable name or a constant.

```
T:  NEW
T:  10 A$="A"
T:  PRINT FRE(D)
R:  7153
T:  10 A$="AB"
T:  PRINT FRE(D)
R:  7152
```

Each character takes one byte.

B. POS(D)

The POS(D) function is used to determine the next position available to print a character; that is, the location of the cursor. The argument is a dummy one and may be alphabetic or numeric.

Exercise: Determine the location of the cursor after a PRINT statement.

```
T:  NEW
T:  10 PRINT A,B,C,POS(D)
T:  20 PRINT A;B;C;POS(D)
T:  RUN
R:  0          0          0          30
R:  0  0  0  9
R:  READY.
```

The numbers 30 and 9 are the next available positions to print characters. Notice that the number of commas and semicolons determines the position of the cursor. The field width is 10 for each variable when using a comma as a separator. The field width for those variables separated by a semi-colon depends on the length of the value in the variable. Two blanks are always inserted between each value separated by a semi-colon.

TRY

```
T:  5 A=1234
T:  RUN
```

What is the result?

Exercise: Check each position of the cursor.

```
T:  NEW
T:  10 FOR I=1 TO 39
T:  20 PRINT SPC(I);"X";POS(D)
T:  30 A=TI+60
T:  40 IF A>TI GO TO 40
```

The TI+60 is a delay of 1 second to allow you to see the action better.

```
T:  50 NEXT I
T:  RUN
```

What is the result? How many print positions per line are there?

There are 40 print positions in each line. POS numbers them 0 to 39.

VIII. BITS AND PIECES

What follows is a collection of items which do not seem to fit any category. It seems like useful information to know, so it has been included.

A. Subroutine Nest Depth

When one subroutine calls another, the subroutines are said to be nested. The maximum nest depth of subroutines is 26. In order to check this, run the following program.

Exercise: Check the maximum subroutine nest depth.

```
T: NEW
T: 5 I=-1
T: 10 I=I+1
T: 20 GO SUB 10
T: RUN
R: ?OUT OF MEMORY ERROR IN 10
T: PRINT I
R: 25
```

The maximum depth of subroutine nesting that is allowed on the PET is 25.

B. Multiple Equalities

The statement A=B=C always sets A=0 regardless of the value of B or C. B or C is not affected by this statement.

Exercise: Multiple equalities.

```
T: NEW
T: A=1:B=2:C=3
T: A=B=C
T: PRINT A;B;C
R:      0     2     3
```

C. Integer Variables

The integer variable I% may be used in place of the INT function since the results are the same.

Exercise: Show that I% may be used in place of the INT function.

```
T: 10 I=INT(5/2+3)
T: 20 I%=5/2+3
T: 30 PRINT I;I%
R: RUN
R: 5  5
```

The use of the integer variable saves typing, saves memory, and simplifies the equation.

D. NEXT Error

Leaving the NEXT out of a loop can cause an error condition in which no error message is generated.

Example:

```
10 FOR I=1 TO 10
.
.
.
90 FOR I=1 TO 5
.
.
.
200 NEXT I
```

By leaving out the NEXT before statement 90 will cause this loop to repeat from statement 90, not from statement 10.

E. Hanging the System

You can hang the system if when loading a program from cassette you hit the 'BREAK' key when the program is partially loaded and then you do a LIST.

Another way to hang the system is to use INPUT# and read past the EOT.

F. Illegal Calculator Mode Statements

INPUT and GET statements are illegal in calculator mode. READ is permitted but you must have a matching DATA statement in memory.

Exercise: Demonstrate statements that are illegal in calculator mode.

```
T:  NEW
T:  INPUT A
R:  ?ILLEGAL DIRECT ERROR
T:  GET A$
R:  ?ILLEGAL DIRECT ERROR
T:  READ A
R:  ?OUT OF DATA ERROR
T:  10 DATA 5
T:  READ A:PRINT A
R:  5
T:  10 DATA X
T:  READ A:PRINT A
R:  ?SYNTAX ERROR IN 10
T:  READ A$:PRINT A$
R:  X
```

The ? is not a valid substitute for the PRINT # statement.

Exercise: Show the use of the ? when used as a substitute for the PRINT.

```
T:  NEW
T:  10 PRINT "PRINT"
T:  20 ? "?"
T:  30 OPEN 1,1,2, "TEST"
T:  40 FOR I=1TO 50
T:  50 PRINT #1,I
T:  60 ?#1,I
T:  70 NEXT I
T:  RUN
R:  PRINT
R:  ?
R:  PRESS PLAY & RECORD ON TAPE #1
R:  OK
R:  ?SYNTAX ERROR IN 60
R:  READY.

T:  60
T:  RUN
R:  PRINT
R:  ?
R:  PRESS PLAY & RECORD ON TAPE #1
R:  OK
R:  READY.
```

As you can see the ?# is not a substiture for PRINT #.

IX. PROGRAMS

A. Blinking Hearts

This program randomly adds hearts to the screen while the display is turned off. After hearts are added, the screen is turned back on for about one second.

```
10 PRINT CHR$(147)
20 POKE 59409,52
30 T=TI+60
40 A=INT(RND(1)*1000)+32768
50 POKE A,83
60 IF T > TI GO TO 40
70 POKE 59409,60
80 T=TI+60
90 IF T > TI GO TO 90
100 GO TO 20
```

Statement 20 turns the screen off while statement 70 turns it back on. If you hit the break key while the screen is blank, just type in POKE 59409,60 to turn it back on. Even though you do not see what you are typing, the system will still take the statement. Notice that there is no "snow" on this display.

B. Number Conversion

This program can be used to convert numbers from decimal to hex, from hex to decimal, from octal to decimal, and from decimal to octal. It may be used by beginners to help them learn new number systems. It may also be used to help convert machine language op codes to decimal to be used with POKE statements.

```
1 PRINT"THIS PROGRAM WILL CONVERT NUMBERS"
2 PRINT"FROM DECIMAL TO HEX;FROM HEX TO"
3 PRINT"DECIMAL;FROM DECIMAL TO OCTAL;"
4 PRINT"AND FROM OCTAL TO DECIMAL. AT THE"
5 PRINT"PROMPT ENTER THE LETTER THAT"
6 PRINT"YOU WANT TO CONVERT FROM AND THE"
7 PRINT"FIRST LETTER YOU WANT TO CONVERT TO."
8 PRINT"THAT IS,DH MEANS YOU WANT TO CONVERT FROM
  DECIMAL TO HEX."
10 PRINT
11 PRINT"ENTER THE CONVERSION YOU WANT"
12 PRINT
13 INPUT N$
14 PRINT
20 IF N$ <> "OH"GO TO 30
22 GOSUB 600
23 PRINT
25 PRINT D;"= ";HS;" HEX"
26 H$=""
27 GO TO 10
30 IF N$ <> "HD" GO TO 40
```

```
32 GOSUB 900
33 PRINT
35 PRINT B;"= ";N$;" HEX"
36 N$=""
37 B=0
38 GO TO 10
40 IF N$ <> "OD" GO TO 50
42 GOSUB 1190
44 PRINT B;"= ";N$;" OCTAL"
45 PRINT
46 GO TO 10
50 IF N$ <> "DO" GO TO 10
55 GOSUB 1050
60 PRINT D;" =";H$;" OCTAL"
65 PRINT
70 GO TO 10
600 PRINT"INPUT THE NUMBER YOU WANTED CONVERTED TO HEX"
601 PRINT
602 INPUT N
605 D=N
611 IF N <> 0 GO TO 620
612 H$="0"
613 RETURN
620 I%=N/16
630 IF I%=0 GO TO 800
640 I=N-(I%*16)
650 IF I <> 0 GO TO 690
660 H$="0"+H$
670 N=I%
680 GO TO 620
690 H$=MID$("123456789ABCDEF",I,1)+H$
700 N=I%
710 GO TO 620
800 I=N-(I%*16)
810 H$=MID$("123456789ABCDEF",I,1)+H$
820 RETURN
830 N=I%
840 GO TO 620
900 PRINT"ENTER THE HEX NUMBER YOU WANT CONVERTED"
905 PRINT
910 INPUT N$
914 FOR J=1 TO LEN(N$)
915 FOR I=1 TO 16
916 IF MID$(N$,J,1)=MID$("0123456789ABCDEF",I,1) GO TO 920
917 NEXT I
918 GO TO 925
920 NEXT J
922 GO TO 930
925 PRINT"INVALID ENTRY":PRINT:GO TO 910
930 J=-1
940 FOR I=LEN(N$)TO 1 STEP-1
```

```
950 J=J+1
960 A$(J)=MID$(N$,I,1)
970 FOR M=1 TO 16
980 IF A$(J)=MID$("0123456789ABCDEF",M,1) GO TO 1000
990 NEXT M
1000 A(J)=M-1
1010 B=B+A(J)*16↑J
1020 NEXT I
1030 RETURN
1050 PRINT"ENTER THE DECIMAL NUMBER YOU WANT CONVERTED
     TO OCTAL"
1055 PRINT
1060 INPUT N
1065 H$=""
1066 D=N
1070 I%=N/8
1080 I=N-(I%*8)
1090 IF I%=0 GO TO 1130
1100 H%=STR$(I)+H$
1110 N=I%
1120 GO TO 1070
1130 H$=STR$(I)+H$
1140 RETURN
1190 PRINT"ENTER THE OCTAL NUMBER YOU WANT CONVERTED"
1195 PRINT
1200 INPUT N$
1202 FOR I=1 TO LEN(N$)
1203 FOR J=1 TO 8
1204 IF MID$(N$,I,1)=MID$("01234567",J,1) GO TO 1207
1205 NEXT J
1206 GO TO 1209
1207 NEXT I
1208 GO TO 1215
1209 PRINT"INVALID ENTRY":PRINT:GO TO 120
1215 B=0
1220 J=-1
1230 FOR I=LEN(N$) TO 1 STEP-1
1240 J=J+1
1250 A$(J)=MID$(N$,I,1)
1310 B=B+VAL(A$(J))*8↑J
1320 NEXT I
1330 RETURN
```

X. TV HARDWARE DISPLAY PROBLEMS

There are two common problems with the PET TV display, overscan and jitter. The overscan problem is relatively easy to correct. Overscan is when the information on the top and bottom lines of the display are at least partially obscured by the cabinet.

To correct overscan, carefully remove the three screws that hold the rear panel on the TV display unit. Use caution since there is high voltage in this cabinet. On the circuit board, near the rear edge, in the middle of the board, is a small trimmer. The trimmer has a slot in it to allow a small screw driver to rotate the trimmer for adjustment. Be careful to touch only the trimmer with your screw driver. If you don't have experience around high voltage electronic equipment, have your TV repairman make the adjustment. Rotating the slot on the trimmer counter-clockwise reduces the size of the display. Rotating the slot clockwise increases the size of the display. The adjustment is very rapid; i.e., a small change in the position of the trimmer makes a great difference in the size of the picture.

The TV jitter problem is a little harder to correct. A common cause of jitter of the characters on the TV screen is overheating of a voltage regulator in the TV unit.

To determine if an overheating regulator is the cause of the character jitters, get an aresol can of freon (camera lens cleaners which remove dust with a puff of air are often freon based). What you need is small blasts of cold air. Run your PET until jitters occur. Carefully remove the back panel from the TV unit. Use caution around the high voltages. Spray a small amount of the cold air on the regulator at the base of the tall heat sink on the left (viewed from back) side of the TV unit. If the regulator is really overheating, each time you cool it, the characters should stop jittering. When the regulator heats up again, the jitters should return.

There are several ways to correct the problem. Most of these solutions probably void the warrantee. All of them require extreme caution around the high voltages.

1. If your PET is still under warrantee (or even if it is not), return it to Commodore for correction.

2. Remove the back panel from the TV unit (see above) to increase the airflow past the regulator and heat sink.

3. Increase the inflow over the regulator heat sink with an internal or external fan.

4. Increase heat transfer away from the regulator by putting heat transfer grease between the regulator and the heat sink.

5. Increase the heat transfer to the air by replacing the heat sink with a bigger or more efficient (finned) unit.

6. Replace the regulator. Be sure the new regulator is a prime unit. A surplus one might be a temperature fallout with the same sort of problem. Early models of PET use the LM340-12 as the regulator.

7. Check the 110V AC line. It may be high. A variable transformer to reduce 120+VAC might keep the regulator within temperature limits. Unless you have the transformer for other purposes, this is an expensive solution.

If you decide to try options 4, 5, or 6, you probably invalidate your warrantee. With these options you will need to remove the circuit board from the TV cabinet. Early models used a compression type of stand-off. Squeeze the stand-off above the board and slip the board up over the stand-off. The stand-offs on the front of the circuit board can be reached easier from the underside of the keyboard.

XI. SAVING AND UTILIZING MEMORY

The best way to utilize memory for large codes is to structure the program in such a way that you can overlay portions as described in Section VI.C.

Some of the methods to save memory do not foster good programming habits and should only be used when you are trying to squeeze every byte from memory.

You can get one byte for each blank you squeeze out of a line. You get at least one byte for the variable you leave off the NEXT statement. You also get a byte if you leave the last quote (") off a print statement. Small statement numbers save memory when used in a GO TO statement. One of the best memory savers (4 bytes/statement/line) is to put multiple statements on a line. Keep variable names to one or two characters, since each character requires one byte.

Using the question mark (?) for the PRINT command does save typing but does not save any memory space.

Using the READ with DATA statements uses more memory than reading the data from cassette. However, there is a trade off in speed and convenience (the cassette is much slower).

XII. UPPER AND LOWER CASE CHARACTERS

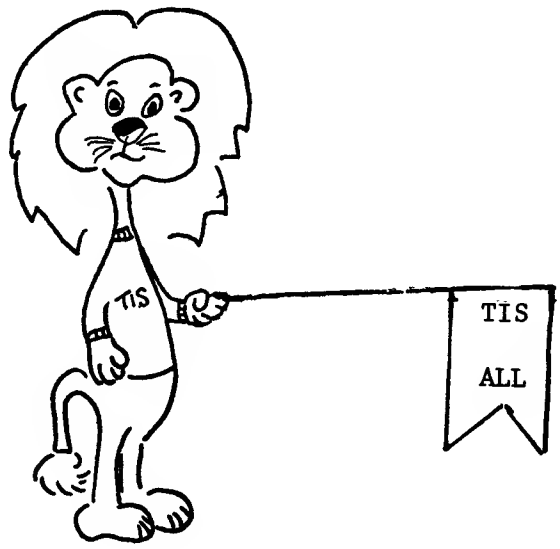
Your PET has two character sets it can display. The "standard" set displays upper case letters, numbers, special characters, and graphical symbols. The "alternate" set displays upper case letters, numbers, special characters, and lower case letters. You cannot display both lower case letters and graphical symbols at the same time. When you turn power on, the standard (graphical) character set is automatically selected. You can tell your PET which character set you want to use by setting one of two specific values (12 or 14) in a particular memory location (59468).

T: POKE 59468,14

Try some shifted characters.

T: POKE 59468,12

Try the same shifted characters.



Notes

zie blz 5-5.

komma 1 naar links ⇒ ×2^-1

komma 1 naar rechts ⇒ ×2^1

dus $a_0 . a_1 a_2 \ldots a_n a_{n+1} \ldots = 0, a_1 a_2 \ldots a_n a_{n+1} \ldots \times 2^n$.

bv. $1011,1101 = 0,10111101 \times 2^4$

analoog $0,0001\,1101 = 0,1101 \times 2^{-3}$

Vb $-3\frac{2}{10} = -3,5$.

$3 \rightarrow 11$

$5 \rightarrow 101$

$-3,5 \rightarrow -11,101 = -0,11101 \times 2^2$

dus 176 ≙ 128 + 32 + 16 10000000

177 11101000

170 t/m 175 00000000

190 10000000 vanwege -